Explicação Código - Trabalho *Conectividade em Sistemas Ciberfísicos*
Equipe: Fernanda Giacobbo, Júlia Melo e Pedro Serafim.

Explicação código <u>Servidor:</u>

1. Importação de Módulos

```
import socket as sock
import threading
```

- `*socket*`: Permite a criação e gerenciamento de conexões de rede. Aqui foi renomeado para `sock` para simplificar o uso.
- `*threading*`: Permite rodar múltiplas tarefas simultaneamente, importante para que o servidor gerencie vários clientes ao mesmo tempo.

---

2. Lista de Clientes

```
clients = []
```

- `*clients*`: Uma lista global que armazena todos os sockets dos clientes conectados ao servidor. É usada para gerenciar a comunicação entre eles.

---

3. Função `broadcast`

```
def broadcast(message, client_socket):
    for client in clients:
        if client != client_socket:
            try:
                client.send(message)
            except:
                client.close()
                clients.remove(client)
```

- Objetivo: Envia uma mensagem para todos os clientes conectados, exceto o que enviou a mensagem original.
- *Parâmetros*:
  - `*message*`: Mensagem a ser enviada (em formato de bytes).
  - `*client_socket*`: O socket do cliente que enviou a mensagem.
- *Funcionamento*:
  1. Percorre a lista `clients`.
  2. Verifica se o cliente atual é diferente do remetente (`client_socket`).
  3. Tenta enviar a mensagem para o cliente.
  4. Se houver erro (cliente desconectado), o socket do cliente é fechado e removido da lista.

—

4. Função `receber_dados`

```
def receber_dados(sock_conn, endereco):
   nome = sock_conn.recv(50).decode()
   print(f"Conexão com sucesso com {nome} : {endereco}")
   broadcast(f"{nome} entrou no chat.".encode('utf-8'), sock_conn)
```

- *Objetivo*: Gerencia a comunicação entre o servidor e um cliente específico.
- *Parâmetros*:
  - `*sock_conn*`: O socket do cliente conectado.
  - `*endereco*`: O endereço IP e porta do cliente.
- *Etapas:*
  1. *Recebe o nome do cliente:*
     - `recv(50)`: Lê até 50 bytes enviados pelo cliente (o nome).
     - `.decode()`: Converte os bytes recebidos em string.
  2. *Confirma conexão no servidor*:
     - `print`: Exibe no servidor que o cliente se conectou.
  3. *Notifica outros clientes*:
     - `broadcast`: Envia uma mensagem informando que o cliente entrou no chat.

Loop de Mensagens:

```
   while True:
      try:
```

```
        mensagem = sock_conn.recv(1024).decode()
        if mensagem.lower() == 'sair':
            sock_conn.close()
            clients.remove(sock_conn)
            broadcast(f"{nome} saiu do chat.".encode('utf-8'), sock_conn)
            break
        else:
            print(f"{nome} >> {mensagem}")
            broadcast(f"{nome} >> {mensagem}".encode('utf-8'), sock_conn)
```

- **Funcionamento**:
  1. *Recebe mensagem*:
     - `recv(1024)`: Recebe até 1024 bytes de dados enviados pelo cliente.
     - `.decode()`: Converte para string.
  2. *Verifica saída*:
     - Se a mensagem for `'sair'`, o cliente é desconectado:
       - O socket é fechado.
       - Removido da lista `clients`.
       - Todos os clientes são notificados.
       - Sai do loop.
  3. *Mensagem normal*:
     - Imprime no servidor.
     - Repassa para todos os outros clientes via `broadcast`.
  4. *Tratamento de erros*:
     - Em caso de erro (como cliente desconectado inesperadamente):
       - Fecha o socket.
       - Remove-o da lista `clients`.
       - Notifica os outros clientes.

---

5. Função `main`

```
def main():
    HOST = '192.168.15.2'
    PORTA = 8888
    socket_server = sock.socket(sock.AF_INET, sock.SOCK_STREAM)
    socket_server.bind((HOST, PORTA))
    socket_server.listen()
    print(f"O servidor {HOST}:{PORTA} está aguardando conexões...")
```

```

- **Configuração do Servidor**:
  1. Define o endereço IP (`HOST`) e a porta (`PORTA`).
  2. Cria um socket:
     - `sock.AF_INET`: Usa IPv4.
     - `sock.SOCK_STREAM`: Usa o protocolo TCP.
  3. Associa o socket ao endereço e porta (`bind`).
  4. Habilita o servidor para aceitar conexões (`listen`).
  5. Exibe uma mensagem indicando que o servidor está pronto.

---

## Loop Principal

```
    while True:
      sock_conn, ender = socket_server.accept()
      clients.append(sock_conn)
      thread_cliente = threading.Thread(target=receber_dados, args=(sock_conn, ender))
      thread_cliente.start()
```

- **Funcionamento:**
  1. *Aguarda conexões:*
     - `accept()`: Aceita um cliente.
     - Retorna:
       - `sock_conn`: Socket do cliente.
       - `ender`: Endereço IP e porta do cliente.
  2. *Armazena o cliente*:
     - Adiciona o socket na lista `clients`.
  3. *Cria uma thread para o cliente:*
     - `threading.Thread`: Cria uma nova thread.
       - `target`: Função que será executada (`receber_dados`).
       - `args`: Argumentos da função.
  4. *Inicia a thread:*
     - `start()`: Permite que o cliente seja gerenciado de forma independente.

---
```

6. Bloco Principal

```
if __name__ == "__main__":
    main()
```

- **Verificação padrão:**
  - Garante que o código no bloco `main` será executado apenas quando o arquivo for executado diretamente (e não importado como módulo).

Explicação código <u>Cliente:</u>

1. Importação de Bibliotecas

```
import socket as sock
import threading
```

- `socket`: Biblioteca padrão para comunicação em rede. Permite criar e gerenciar conexões de rede.
  - Aqui, renomeamos o módulo como `sock` usando o `as`. Isso é útil para evitar conflitos de nomes ou tornar o código mais curto.
- `threading`: Biblioteca para trabalhar com threads, que permitem a execução de múltiplas tarefas simultaneamente. Neste código, é usada para escutar mensagens do servidor enquanto o usuário pode enviar mensagens.

---

2. Função `receive_messages`

```
def receive_messages(client_socket):
    while True:
        try:
            message = client_socket.recv(1024).decode('utf-8')
            if message:
                print(message)
            else:
                client_socket.close()
                break
```

```
    except:
        client_socket.close()
        break
```

Essa função é responsável por escutar e exibir as mensagens recebidas do servidor. É executada em um thread separado para não bloquear o fluxo principal.

Detalhes:
- `client_socket`: Socket do cliente usado para comunicação com o servidor.
- `recv(1024)`:
  - Método do socket que recebe dados da conexão.
  - O argumento `1024` indica o tamanho máximo de dados (em bytes) a ser recebido em cada chamada.
- `decode('utf-8')`:
  - Converte os dados recebidos (em formato de bytes) para uma string usando a codificação UTF-8.
- Verificação de `message`:
  - Se `message` não for vazio, é exibido na tela com `print()`.
  - Caso contrário (conexão encerrada pelo servidor), o socket é fechado e o loop é interrompido.
- `except`:
  - Captura qualquer exceção (como falhas na conexão) e fecha o socket antes de sair.

---

3. Função `main`

```
def main():
    HOST = '192.168.15.2'
    PORTA = 8888
    socket_cliente = sock.socket(sock.AF_INET, sock.SOCK_STREAM)
    socket_cliente.connect((HOST, PORTA))
```

Configuração inicial:
- `HOST`: Endereço IP do servidor.
- `PORTA`: Porta usada para a conexão.
- `socket_cliente = sock.socket(sock.AF_INET, sock.SOCK_STREAM)`:
  - Cria um socket para conexão TCP/IP (Protocolo de Transporte baseado em fluxo de dados).

  - `AF_INET`: Especifica que estamos usando IPv4.
  - `SOCK_STREAM`: Indica que o protocolo usado será o TCP.
- `connect((HOST, PORTA))`:
  - Conecta o socket do cliente ao servidor no endereço e porta especificados.

Exibindo informações iniciais e enviando o nome do usuário:
```
    print(5"" + "INICIANDO CHAT" + 5"")
   nome = input("Informe seu nome para entrar no chat:\n")
   socket_cliente.sendall(nome.encode())
```
- Exibe uma mensagem de boas-vindas.
- Solicita ao usuário seu nome para identificação no chat.
- `sendall(nome.encode())`:
  - Envia o nome do usuário ao servidor.
  - `encode()`: Converte a string para bytes antes de enviar.

Criando e iniciando o thread para receber mensagens:
```
    thread = threading.Thread(target=receive_messages, args=(socket_cliente,))
    thread.start()
```
- `threading.Thread()`:
  - Cria um novo thread que executa a função `receive_messages`, passando o socket do cliente como argumento.
- `start()`:
  - Inicia a execução do thread, permitindo que ele funcione paralelamente ao código principal.

Loop principal para envio de mensagens:
```
    while True:
      mensagem = input('')
      if mensagem.lower() == 'sair':
         socket_cliente.sendall(mensagem.encode('utf-8'))
```

```
            socket_cliente.close()
            break
        else:
            socket_cliente.sendall(mensagem.encode('utf-8'))
```
- `mensagem = input('')`:
  - Lê a mensagem digitada pelo usuário.
- `mensagem.lower() == 'sair'`:
  - Verifica se o usuário digitou "sair" (independente de maiúsculas/minúsculas). Se sim:
    - Envia a mensagem "sair" ao servidor.
    - Fecha o socket do cliente.
    - Sai do loop.
- `else`:
  - Envia a mensagem ao servidor usando `sendall()`.

---

4. Bloco de execução principal

```
if __name__ == "__main__":
    main()
```
- Verifica se o script está sendo executado diretamente (e não importado como módulo).
- Se sim, chama a função `main()` para iniciar o cliente.